Parallelizing the Slant Stack Transform with CUDA

Dustin Barnes
Computer Science and Engineering
University of Nevada, Reno
Reno, NV, USA
dkbarnes@nevada.unr.edu

Andrew McIntyre
Computer Science and Engineering
University of Nevada, Reno
Reno, NV, USA
amcintyre@nevada.unr.edu

Sui Cheung
Computer Science and Engineering
University of Nevada, Reno
Reno, NV, USA
scheung@nevada.unr.edu

John Louie
Nevada Seismological Laboratory
University of Nevada, Reno
Reno, NV, USA
louie@seismo.unr.edu

Emily Hand
Computer Science and Engineering
University of Nevada, Reno
Reno, NV, USA
emhand@unr.edu

Frederick C. Harris, Jr.
Computer Science and Engineering
University of Nevada, Reno
Reno, NV, USA
fred.harris@cse.unr.edu

***Abstract*—In geophysics, the slant stack transform is a method used to align signals from different sensors. We focus on the use of the transform within passive refraction microtremor (ReMi) surveys, in order to produce high resolution slowness-frequency plots for use as samples in a machine learning model. Running on a single central processing unit (CPU) thread, this process takes approximately 45 minutes, 99.5% of which consists of the slant stack transform. In order to reduce the time taken to perform the transform, we use NVIDIA CUDA programming model. Using the same CPU, augmented with a GeForce RTX 2080 Ti we were able to reduce this time down to as little as 0.5 seconds.**

***Index Terms*—Parallel Programming, GPU, CUDA, PyCUDA, Refraction Microtremor (ReMi), Seismic Refraction, Slant Stack, Radon Transform, Beamforming Transform, Rayleigh Waves,**

I. Introduction

Refraction Microtremor (ReMi) surveys use geophone-array recordings of the Rayleigh component of ambient, vertically directed seismic ground-vibration noise from passive sources in order to obtain a shear-wave profile, describing the seismic-velocity property of soils and rocks at different depths [1]. ReMi does not require drilling or the use of a seismic source - such as hammers or explosives. ReMi provides a cheap, easy to set up survey method that is effective in urban environments. Although ReMi was originally developed as a means of assessing earthquake safety and construction code compliance for building sites, ReMi has been used for a wide variety of applications such as geologic basin and bedrock analysis, and foundation design for sites such as bridges and wind turbines.

Conducting a ReMi survey require obtaining array recordings, transforming the time- and distance-dependent seismic records into a slowness-frequency (p-f) plot, manually estimating the fundamental-mode Rayleigh dispersion curve, and then manually fitting the selected dispersion points to a 1D shear-velocity versus depth model. This is a labor-intensive process with a mathematically under-determined result, which can vary depending on the individual performing the analysis. In addition, the amount of raw seismic data being recorded for analysis is constantly increasing, beyond the capacity of human analysts to keep up. As such, the ultimate goal of this research is to generate a machine learning model capable of generating a velocity model given a p-f plot.

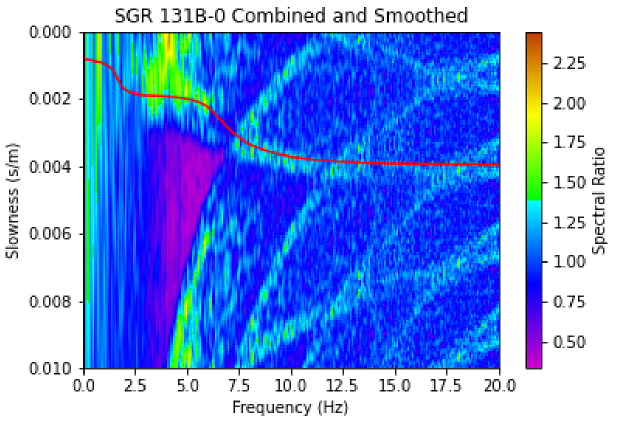

Fig. 1. An example of a generated p-f plot, with the fundamental mode Rayleigh dispersion curve drawn in red. This one was generated using the serial implementation.

Methods of generating p-f plots are designed for human consumption, constrained by the limitations of decades old hardware. Increasing the demands on this algorithm to create samples suitable for consumption by a machine learning model make the time required to generate a batch of samples infeasible for training. The vast majority of this time is the result of performing the slant stack transform, which aligns the signals from the geophone sensors along the time-offset (tau) domain. Figure 1 is an example of visualized waves in geophysics. It is built from the sensor measurements in a linear array (traces). It accumulate samples (intensity) at each point and it is currently in a low resolution grids for program performance purpose.

The remainder of the paper is structured as follows: Section II provides an overview of the ReMi process and seismic imaging techniques. Section III provides a brief background for CUDA. Section IV provides a description of the algorithm which slant stacking utilizes, and different methods of optimizing this for CUDA. Section V provides details on hardware and software used. Section VI describes the outcomes of our study, and Section VII provides conclusion and a discussion on future work, potential improvements, and applications of our work.

II. Background

Several other works have shown the potential for CUDA accelerated computation within geophysics. [2] and [3] are both examples of using graphics processing units (GPU) to accelerate processing, while [4] shows how ray tracing on modern hardware can be used to increase the fidelity of computational models. No work has been done on accelerating the slant stack transform or ReMi surveys, with most research in the field instead investigating different areas in which the technique may be applied.

The remainder of this paper focuses on generating p-f plots and the slant stack transform. Generating a p-f plot consists of several distinct steps. First, the raw signals must be processed. Each trace, consisting of the raw signals from each geophone, must be centered, ensuring that they are measured as variations around a zero-volt response level, and RMS normalization is performed. Next, the signal is transformed into the frequency domain and filtered. At minimum, a bandpass filter is used with the maximum frequency determined by the Nyquist frequency - the highest frequency able to be reliably sampled, based on the sampling rate of the sensors - using a bandpass filter. After frequency domain filtering is performed, the signal for each trace is then transformed back into the time domain.

Once the raw data has been preprocessed, the slant stack transform is applied. This generates a p-τ plot, with axes of p, or slowness (s/m) and τ, or zero offset reflection time, and with each point on the grid representing the intensity of the signal received.

$$\tau = t - p * d \tag{1}$$

$$t = \tau + p * d \tag{2}$$

As shown in equation (1), τ is calculated by taking the distance (d) of the current sensor from the first sensor in a linear array and multiplying it by the current slowness (p) to align the signal from a distant geophone to that of the first. The intensity of the signal is the added to the current grid point. When generating the p-τ plot, every point in the plot is iterated over, and the time in the trace is calculated using Eq. (2). As the calculated trace time is unlikely to align exactly with a sampled point on the trace, linear interpolation is performed between the two nearest values.

In the case where a time is outside the bounds of the trace, either with negative time or beyond the sampled time frame, no intensity is added. In addition to creating a plot from slowness $p = 0$ to p_{max}, a p-τ plot is also created ranging from $-p_{max}$ to $p = 0$. Thus, the plot ranging from [0,p_{max}] captures waves travelling along the array from the first sensor to last, and the plot ranging from [-p_max, 0] captures waves travelling along the array from the last sensor to the first. These are referred to as the forward and reverse plots respectively.

Once the forward and reverse p-τ plots have been calculated, the reverse plot is inverted along the slowness axis, and added to the forward plot, creating the combined plot. A final Fourier transform is applied to all three plots, transforming them into the frequency domain along the x axis. Additional work may be done in order to visualize the data in different ways.

III. CUDA

The graphical processing unit (GPU) has become a popular device for creating a high performance parallel computing platform for a relatively low cost. Each GPU contains a large number of processing cores and each core can create many threads that can all be executed in parallel. As a result, many different types applications utilize GPUs to solve computationally expensive problems and have been successful in increasing the performance. NVIDIA provides their Compute Unified Device Architecture (CUDA) programming model in order for new and existing applications to be executed on the GPU. CUDA extends the standard C/C++ language to give direct access to the instructions and memory management for parallel computation on NVIDIA GPUs

Starting with the G80 series, NVidia unveiled a new architecture called the Compute Unified Device Architecture (CUDA) to ease the struggles of programmers attempting to harness the GPU's computing power [5]. While the new architect did not change the pipeline for graphics programmers, it did unify the processing architecture underlying the whole pipeline. Vertex and fragment processors were replaced with groups of Thread Processors (CUDA Cores) called Streaming Multiprocessors (SM). Initially with the G80 architecture, there were 128 cores grouped into 16 SMs. The Kepler architecture has 2880 coresgrouped into 15 SMs. The Maxwell has SMs with 128 cores and has a varying number of SMs (3-15) depending upon the model of GPU. The Pascal architecture was also 128, but Turing was 64, and Ampre has gone back to 128

In addition to the new architecture, CUDA also included a new programming model which allowed application programmers to harness the data parallelism present on the GPU. The primary abstractions of the programming model are kernels and threads. Each kernel is executed by many threads in parallel; CUDA threads are very lightweight and allow many thousands of threads to be executing on a device at any given time.

CUDA exposes the computational power of the GPU through a C programming model. It additionally provides an API for scheduling multiple streams of execution on the GPU. This allows the hardware scheduler present on CUDA enabled GPUs to more efficiently use all of the compute resources available. When using streams, the scheduler is able to concurrently execute independent tasks.

Changes over the years to CUDA have included shared memory between the host and device as well as kernel calls from kernels.

IV. Approach/Implementation

The slant stack algorithm itself is fairly straightforward. Algorithm 1 describes the basic serial implementation of the transform, with Algorithm 4 describing the processes used to interpolate the intensity of a signal on a trace, given the ideal time. This process is identical for both the serial and parallel algorithms, however it does require a significant amount of computation. Our test case, generating a 1648x1648 plot from a stream with 15 traces, requires 147 million executions of Algorithm 4.

Algorithm 1: Serial Slant Stack Transform

```
Create Output Plot
for trace do
    for tau do
        for $p$ do
            $t_{ideal} = \tau + d * p$
            Accumulate on plot // Call Alg 4
        end
    end
end
```

For our parallel implementation, two algorithms are described. Algorithm 2 performs a CUDA call on each trace in a stream, limiting the amount of memory required at the cost of more frequent CUDA calls. Algorithm 3 sends the full stream to the GPU, and processes it in a single CUDA call, but there must be sufficient VRAM available on the device for the output array, all traces, as well as memory available for computation.

When Algorithm 2 is executed, a trace is sent to the GPU and the result of each point in the output array is calculated and summed to the same global array. Once all traces have been called, the results are transferred back to the host. This method simplifies the process - each trace modifies each point of the output array once, and so it is guaranteed that there will be no race conditions present when calculating a single trace - as well as reducing the amount of memory required for computation. For our test case, the output array requires approximately 1.28 GB of VRAM to store, with additional memory needed for the trace itself. Some additional cost is also incurred due to the more frequent communication required between the device and host, though it is not significant unless generating very small plots.

Algorithm 3 performs a batch computation of all traces. Similar to Algorithm 2, a global output array is used to store the results. However consideration must be given to potential data races, as multiple threads may attempt to modify the same memory locations. In this case, each trace within the stream will modify each element of the global output array once - resulting in 15 increments of every value. This requires additional checks to ensure that no two threads are operating on the same location, either via use of atomics, temporary arrays for each trace, or subdividing the output array into several smaller arrays. Additionally, loading the entirety of the stream into VRAM may cause difficulties depending on the size of the survey. For the purposes of this study, we only implement Algorithm 2.

To implement these methods, we use the PyCUDA library, which gives access to NVIDIA's CUDA parallel computation API in python. This allows us to easily parallelize our serial code, which is implemented in python. Additionally, utilizing python allows for easier integration with machine learning models, and better support for reading geophysics data.

Algorithm 2: CUDA Kernel per Trace

```
CPU:

Create Output Plot
for trace do
    CUDA Call
end

KERNEL:

$t_{ideal}$ = threadIdx.x+threadIdx.y*distance
Add to Output Array // Call Alg 4
```

Algorithm 3: CUDA Kernel per Stream

```
CPU:

Create Output Plot
Send all traces to GPU
CUDA Call

KERNEL:

Calculate Trace Time
$t_{ideal}$ = threadIdx.x
    + threadIdx.y*distance
    + threadIdx.z*traceLen
Add to Output Array // Call Alg 4
```

Algorithm 4: Process to find intensity

```
if $abs(t_{ideal}) < t_{max}$ then
    $t_{low} = floor(t_{ideal}/dt)$
    $t_{high} = ceil(t_{ideal}/dt)$
    intensity = $\frac{trace[t_{high}]-trace[t_{low}]}{t_{high}-t_{low}}(t_{ideal} - t_{low})$
    plot[$\tau$][p] += intensity
end
else
    Skip
end
```

V. Hardware and Software Used

CUDA and PyCUDA are used in this project both in the sequential and parallel implementations. As mentioned in Section III, CUDA is parallel computing platform and programming model that was designed for NVIDIA graphics cards. CUDA helps speed up computing application significantly when there is a lot of data that is computed at the same time. CUDA supports languages such as C, C++, Fortran, Python, and Matlab. In this work, we utilized PyCUDA for a python implementation of the slant stack transform implementation [6].

PyCUDA is a Pythonic access method for NVIDIA's CUDA parallel computation API. PyCUDA has C++ style syntax so that it's ease-to-use. It provides automatic error checking where all CUDA calls are translated as Python exceptions. It has a very fast runtime, along with automatically allocating and de-allocating space in the program [7].

A. Hardware used

The hardware described was used for both the sequential and parallel implementations of the code. Since CUDA is limited to use for NVIDIA graphics cards. It is very important to have a workstation that is set up with NVIDIA graphics card(s). We also wanted to have a decent CPU where it can send the CUDA instructions to the NVIDIA graphics card(s) rapidly. For this paper, we used a workstation that has NVIDIA graphics card and a Intel CPU. Here is the essential components of our workstation set up:

- Intel® Core™ i5-4670K Processor
- GeForce RTX 2080 Ti

Intel® Core™ i5-4670K Processor is the 4th generation Intel® Core™ i5 Processors. It has 4 cores and 4 threads, processor base frequency of 3.40GHz, and 6MB of Intel® Smart Cache (which allows all cores to dynamically share access to the last level cache) [8].

The machine was running a Debian version of Linux.

GeForce RTX 2080 Ti was the lastest version of NVIDIA graphics card at the time of this work. A GeForce RTX 2080 Ti was used in the workstation and it is made by EVGA and the model is 11G-P4-2281-KR. A GeForce RTX 2080 Ti has 4352 of NVIDIA CUDA® Cores, 14 Gbps of memory speed, 11 GB GDDR6 VRAM.

B. Software used

In the CUDA implementation, we used software packages from *CUDA 10.2*, *pycuda 2019.1.2* [7], *NVIDIA Driver 3.5*, *numpy 1.18.2* [9], *obspy 1.2.1* [10], [11], and *evodcinv 1.0.0* [12]. ObsPy is a seismology framework for Python, which provides tools necessary to parse common file formats. In this project, ObsPy is exclusively used to read in the trace data.

VI. Results

Implementing the slant stack transform in CUDA allowed for a considerable speed up, while still producing accurate results. Prior to CUDA parallelization, an individual trace would take approximately 400 seconds to to calculate using the CPU described in Section V. This resulted in our test case needing approximately one and a half hours of calculation. After parallization was implemented, a single trace could be processed in approximately 0.01 seconds, with the entire process averaging 0.65 seconds. We observed an average speedup of approximately 10,000 times. This is consistent with the peak performance of the two devices, as the the 2080ti is capable of 14.2 teraflops (TFLOPS), compared to the 13.6 gigaflops (GFLOPS) available when using a single core of on the CPU.

We measured both GPU memory and processory utilization using the NVIDIA System Management Interface. As discussed in Section IV, Algorithm 2 was unable to saturate the GPU, peaking at only 5 percent GPU utilization. Despite this, memory utilization peaked at 40 percent, or 4.4GB. This shows that there is still room for improvement, such as processing multiple traces at once.

Figure 2 shows the CUDA generated plots, with one zoomed in, covering the same region as the serial version shown in Figure 1. The second plot shows the same data, beyond the fundamental mode dispersion curve and extending to the Nyquist frequency of our test case. It's worth noting that these plots are both generated with the same resolution, however the larger one is impractical for use without using GPU acceleration.

Figure 3 shows the runtime for both devices at different problem sizes. The runtime scales linearly with the number of grid points that need to be calculated, with some inconsistencies on the GPU when calculating with very small problem sizes.

VII. Conclusion and Future Work

The findings in this project indicate that the spreading the calculations for the slant stack transform across multiple CUDA cores provides substantial speedup. The drastic increase in the amount of data that the program can produce in a certain amount of time makes the method viable for use in machine learning. By making the slant stack transform a parallel process through CUDA, we were able to achieve close to real time computation.

Although in this work we apply the slant stack transform to ReMi surveys, the transform is not specific to this technique. As such, this can be beneficial to any work which uses slant stacking, particularly if processing a large amount of files, or attempting to generate plots with high resolution.

Future work for this project includes having the calculations be performed through batch processing. This would significantly boost the output of the program, since more data can be generated over a shorter period of time. In addition to the slant stack transform, other parts of the algorithm can be parallelized. For example, the process of transforming and filtering the raw data is currently a serial operation. While the overhead from preprocessing everything in a serial fashion is small, larger inputs may negatively affect the runtime of the program. The Fourier transforms performed on the set of traces

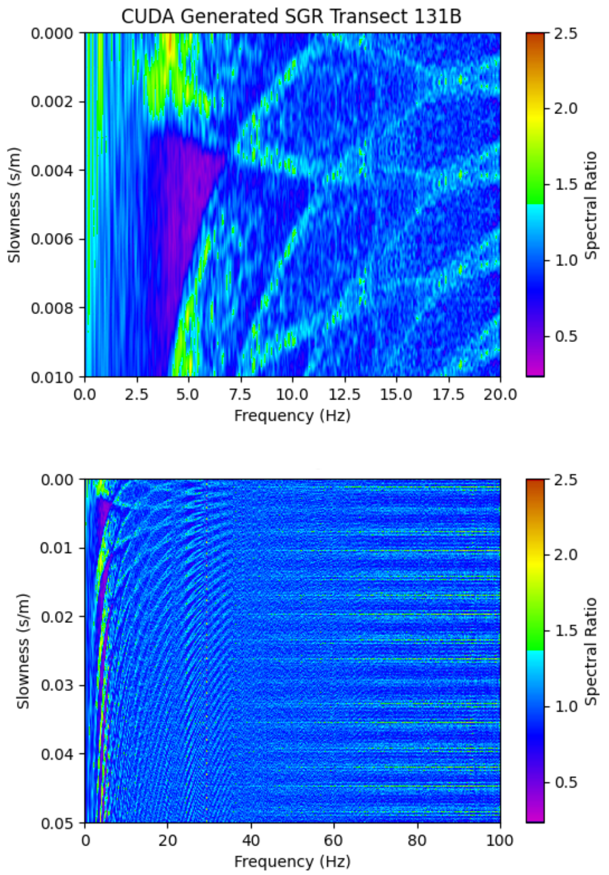

Fig. 2. These p-τ plots were generated using our CUDA implementation. The top plot shows the zoomed in figure, comparable to the one presented in Fig. 1. The bottom plot shows the full figure, extending to the Nyquist frequency of our test case.

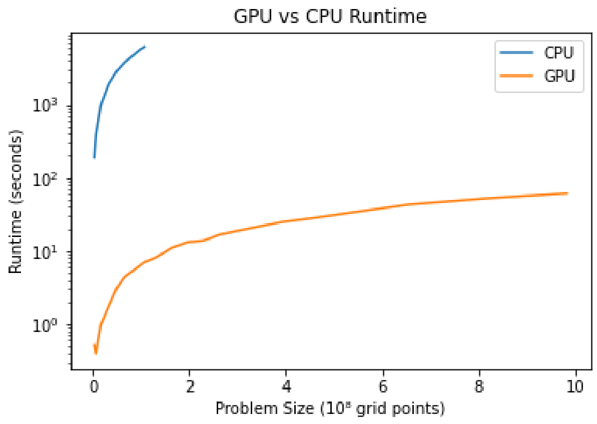

Fig. 3. This is the runtime comparison plot between the Intel® Core™ i5-4670K (CPU) and the GeForce RTX 2080 Ti (GPU).

can also be implemented into CUDA as well, limiting the data that needs to be sent to the GPU to only the raw traces.

References

[1] J. Louie, "Faster, better: Shear-wave velocity to 100 meters depth from refraction microtremor arrays," *Bulletin of the Seismological Society of America*, vol. 91, 04 2001.

[2] Y. Wang, H. Zhou, X. Zhao, Q. Zhang, P. Zhao, X. Yu, and Y. Chen, "CuQ-RTM: A CUDA-based code package for stable and efficient q-compensated reverse time migration," *GEOPHYSICS*, vol. 84, no. 1, pp. F1–F15, Jan. 2019. [Online]. Available: https://doi.org/10.1190/geo2017-0624.1

[3] B. Holt and D. Ernst, "Accelerating geophysics simulation using cuda," *The Journal of Computational Science Education*, vol. 2, pp. 21–27, 12 2011.

[4] M. Sarajaervi and H. Keers, "Ray-based modeling and imaging in viscoelastic media using graphics processing units," *GEOPHYSICS*, vol. 84, no. 5, pp. S425–S436, Sep. 2019. [Online]. Available: https://doi.org/10.1190/geo2018-0510.1

[5] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, p. 40–53, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1365490.1365500

[7] A. Klöckner, N. Pinto, B. Catanzaro, Y. Lee, P. Ivanov, and A. Fasih, "GPU scripting and code generation with PyCUDA," in *GPU Computing Gems Jade Edition*, W. mei W. Hwu, Ed. Elsevier Inc, 2012, pp. 373–385.

[8] Intel Corp., "Intel® core™ i5-4670k processor: 6m cache, up to 3.80 ghz," visited on (11/2/2020). [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/75048/intel-core-i5-4670k-processor-6m-cache-up-to-3-80-ghz.html

[9] NumPy, "NumPy: The fundamental package for scientific computing with python," visited on (11/2/2020). [Online]. Available: https://numpy.org/

[10] The ObsPy Development Team, "ObsPy: A python framework for seismology," visited on (11/2/2020). [Online]. Available: https://github.com/obspy/obspy/wiki

[11] M. Beyreuther, R. Barsch, L. Krischer, T. Megies, Y. Behr, and J. Wassermann, "ObsPy: A Python Toolbox for Seismology," *Seismological Research Letters*, vol. 81, no. 3, pp. 530–533, 05 2010. [Online]. Available: https://doi.org/10.1785/gssrl.81.3.530

[12] K. Luu, "EvoDCinv," visited on (11/2/2020). [Online]. Available: https://github.com/keurfonluu/EvoDCinv